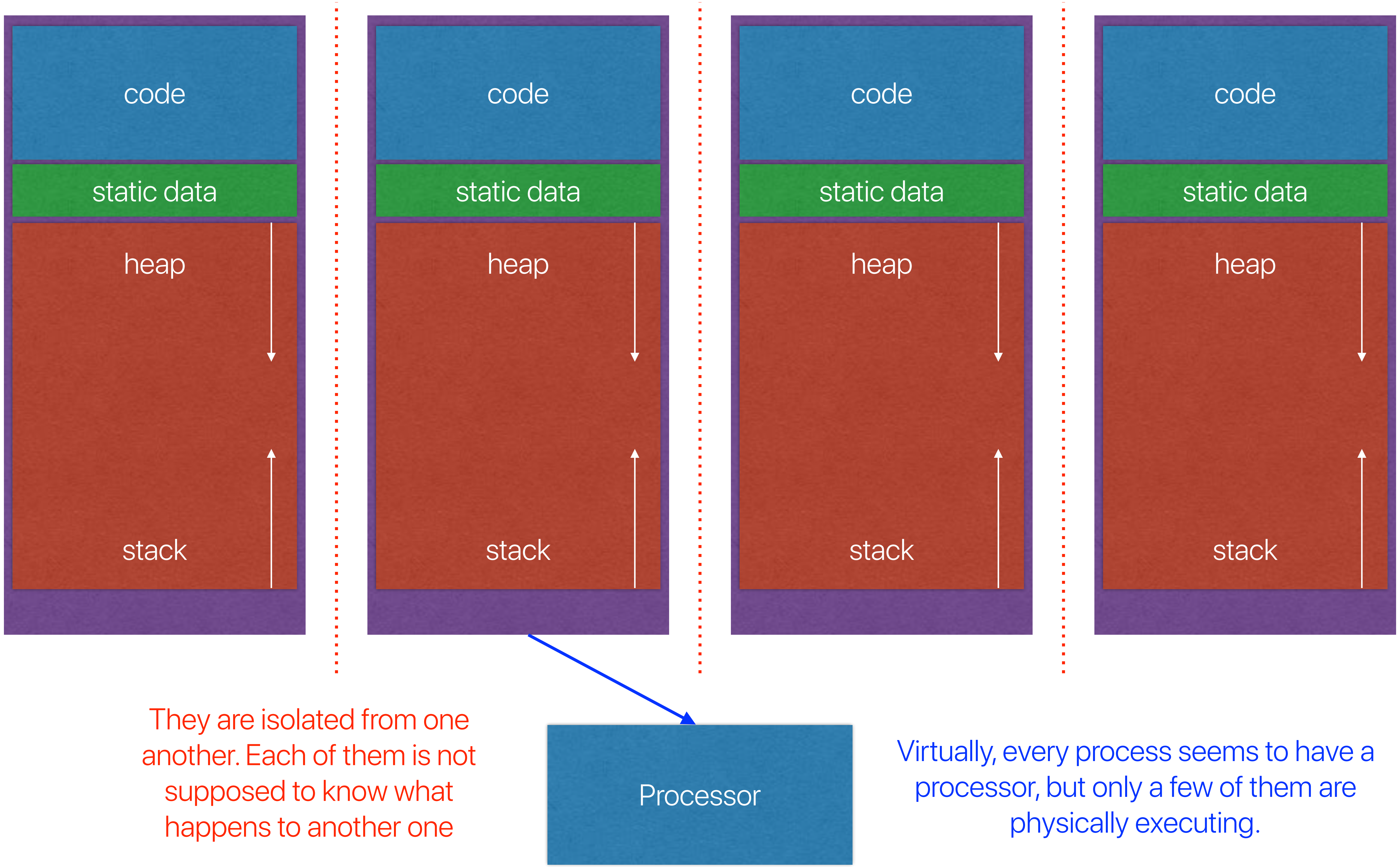
Design philosophy of operating systems (III)

Hung-Wei Tseng

Outline

- The process interface in UNIX
- Mach: A New Kernel Foundation For UNIX Development

Each process has a separate virtual memory space



They are isolated from one another. Each of them is not supposed to know what happens to another one

Virtually, every process seems to have a processor, but only a few of them are physically executing.

Review the first demo

```
[2] 19110
[3] 19111

Process A is using CPU: 1. Value of a is 1052337033.000000 and address of a is 0x601090

Process B is using CPU: 3. Value of a is 1841722078.000000 and address of a is 0x601090

Process C is using CPU: 0. Value of a is 451378955.000000 and address of a is 0x601090

Process D is using CPU: 0. Value of a is 1227583454.000000 and address of a is 0x601090

Process A is using CPU: 1. Value of a is 1052337033.000000 and address of a is 0x601090

Process B is using CPU: 3. Value of a is 1841722078.000000 and address of a is 0x601090

Process C is using CPU: 0. Value of a is 451378955.000000 and address of a is 0x601090

[1]     Done                          ./virtualization A

[2]     Done                          ./virtualization B

[3]     Done                          ./virtualization C

Process D is using CPU: 0. Value of a is 1227583454.000000 and address of a is 0x601090
escal02 [/home/htseng3/courses/CSC501/virtualization] -htseng3-
```

The interface of managing processes

The basic process API of UNIX

- `fork`
- `wait`
- `exec`
- `exit`

`fork()`

- `pid_t fork();`
- `fork` used to create processes (UNIX)
- What does `fork()` do?
  - Creates a **new** address space (for child)
  - **Copies** parent's address space to child's
  - Points kernel resources to the parent's resources (e.g. open files)
  - Inserts child process into ready queue
- `fork()` returns twice
  - Returns the child's PID to the parent
  - Returns "0" to the child

`exit()`

- `void exit(int status)`
- `exit` frees resources and terminates the process
  - Runs an functions registered with atexit
  - Flush and close all open files/streams
  - Releases allocated memory.
  - Remove process from kernel data structures (e.g. queues)
- `status` is passed to parent process
  - By convention, 0 indicates “normal exit”

Starting a new program with `execvp()`

- `int execvp(char *prog, char *argv[])`
- `fork` does not start a new program, just duplicates the current program
- What `execvp` does:
  - Stops the current process
  - Overwrites process' address space for the new program
  - Initializes hardware context and args for the new program
  - Inserts the process into the ready queue
- `execvp` does not create a new process

Why separate `fork()` and `exec()`

- Windows only has `exec`
- Flexibility
- Allows redirection & pipe
  - The shell `forks` a new process whenever user invoke a program
  - After `fork`, the shell can setup any appropriate environment variable to before `exec`
  - The shell can easily redirect the output in shell: a.out > file

Let’s write our own shells

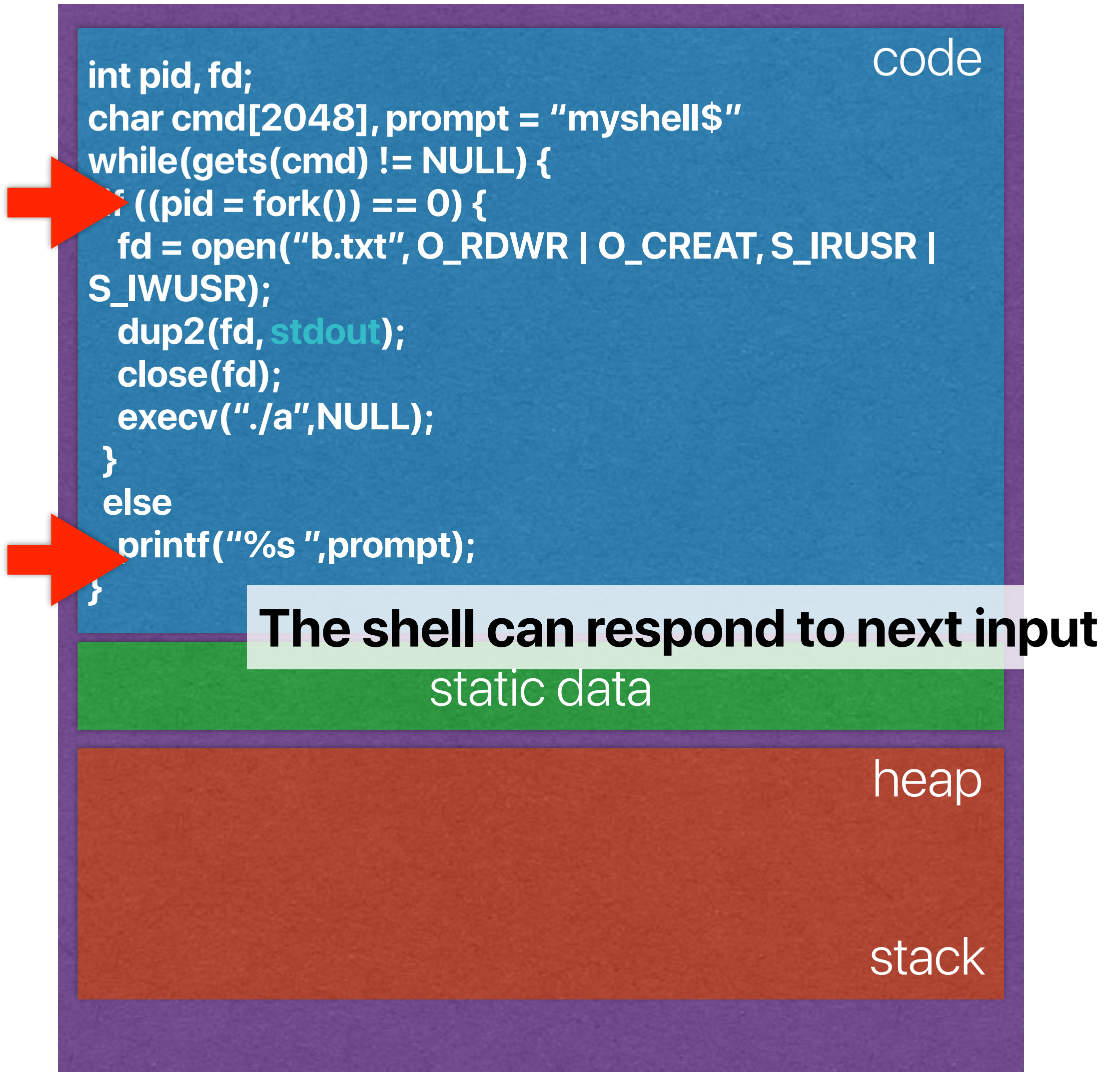
How to implement redirection in shell

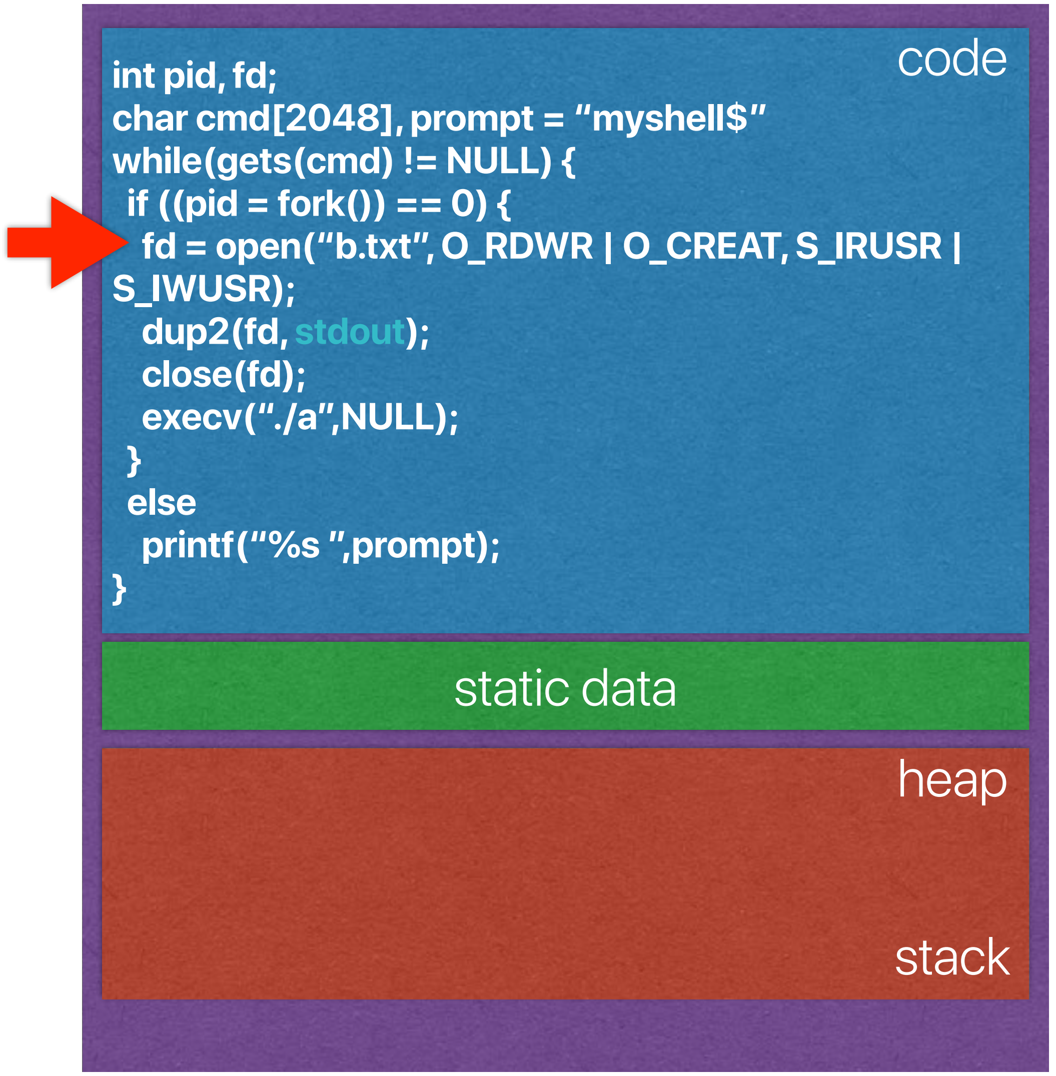
- Say, we want to do ./a > b.txt
- fork
- The forked code opens b.txt
- The forked code dup the file descriptor
- The forked code assigns b.txt to stdin/stdout
- The forked code closes b.txt
- exec("./a", NULL)

How to implement redirection in shell

- Say, we want to do ./a > b.txt
- fork
- The forked code opens b.txt
- The forked code dup the file descriptor to stdin/stdout
- The forked code closes b.txt
- exec("./a", NULL)

Homework for you:
Think about the case when
your `fork` is equivalent to `fork+exec`()





wait()

- `pid_t wait(int *stat)`
- `pid_t waitpid(pid_t pid, int *stat, int opts)`
- `wait` / `waitpid` suspends process until a child process ends
  - `wait` resumes when any child ends
  - `waitpid` resumes with child with pid ends
  - `exit` status info 1 is stored in *stat
  - Returns pid of child that ended, or -1 on error
- Unix requires a corresponding `wait` for every `fork`

Starting a new program with `exec()`

- `int execvp(char *prog, char *argv[])`
- `fork` does not start a new program, just duplicates the current program
- What `exec` does:
  - Stops the current process
  - Overwrites process' address space with a new one for prog
  - Initializes hardware context and args for the new program
  - Inserts the process into the ready queue
- `exec` does not create a new process

How to implement redirection in windows

- Say, we want to do ./a > b.txt
- The shell opens b.txt
- The shell saves stdin/stdout
- The shell assigns b.txt to stdin/stdout
- exec(“./a”, NULL)
- The shell closes b.txt
- The shell restores stdin/stdout

Zombies, Orphans, and Adoption

- Zombie: process that exits but whose parent doesn't call wait
  - Can't be killed normally
  - Resources freed but pid remains in use
- Orphan: Process whose parent has exited before it has
  - Orphans are **adopted** by init process, which calls wait periodically

Mach: A New Kernel Foundation For UNIX Development

Mike Accetta , Robert Baron , William Bolosky , David Golub , Richard Rashid , Avadis Tevanian , Michael Young
Computer Science Department, Carnegie Mellon University

Why "Mach"?

- The hardware is changing
  - Multiprocessors
  - Networked computing

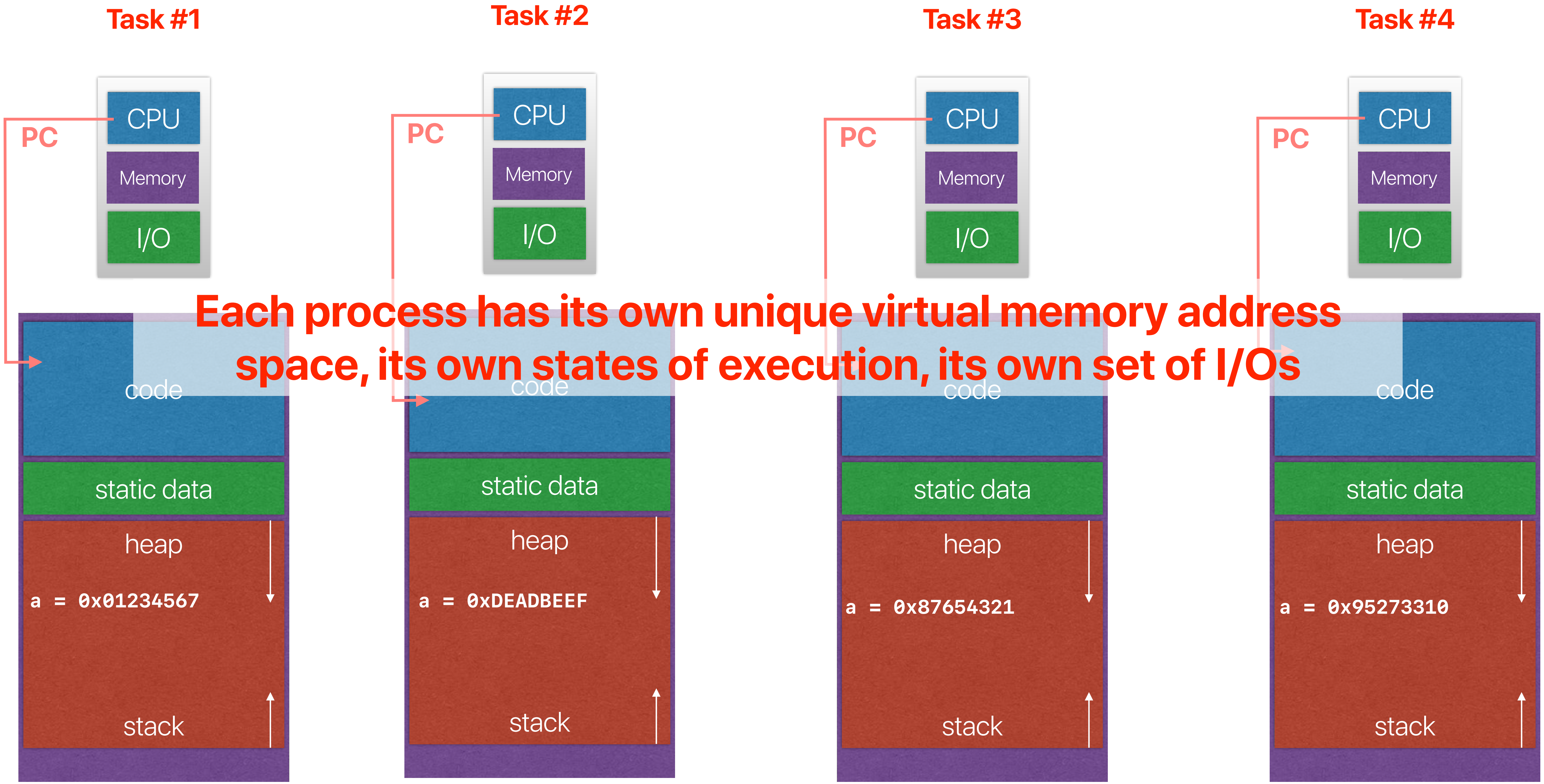
> be built and future development of UNIX-like systems for new architectures can continue. The computing environment for which Mach is targeted spans a wide class of systems, providing basic support for large, general purpose multiprocessors, smaller multiprocessor networks and individual workstations (see

- The software
  - The demand of extending an OS easily
  - Repetitive but confusing mechanisms for similar stuffs

> As the complexity of distributed environments and multiprocessor architectures increases, it becomes increasingly important to return to the original UNIX model of consistent interfaces to system facilities. Moreover, there is a clear need to allow the underlying system to be transparently extended to allow user-state processes to provide services which in the past could only be fully integrated into UNIX by adding code to the operating system kernel.

Make UNIX great again!

Tasks/processes

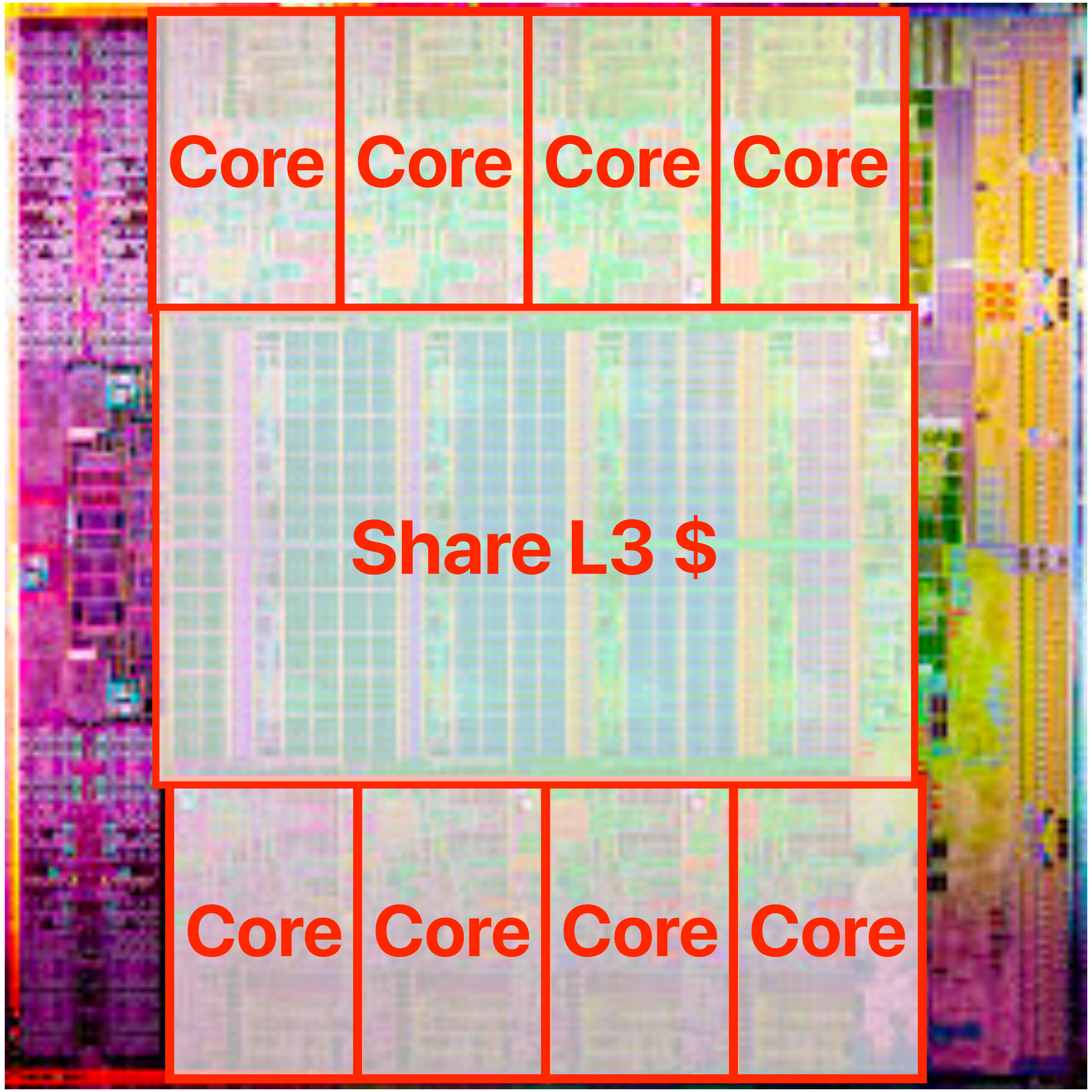


Each process has its own unique virtual memory address space, its own states of execution, its own set of I/Os

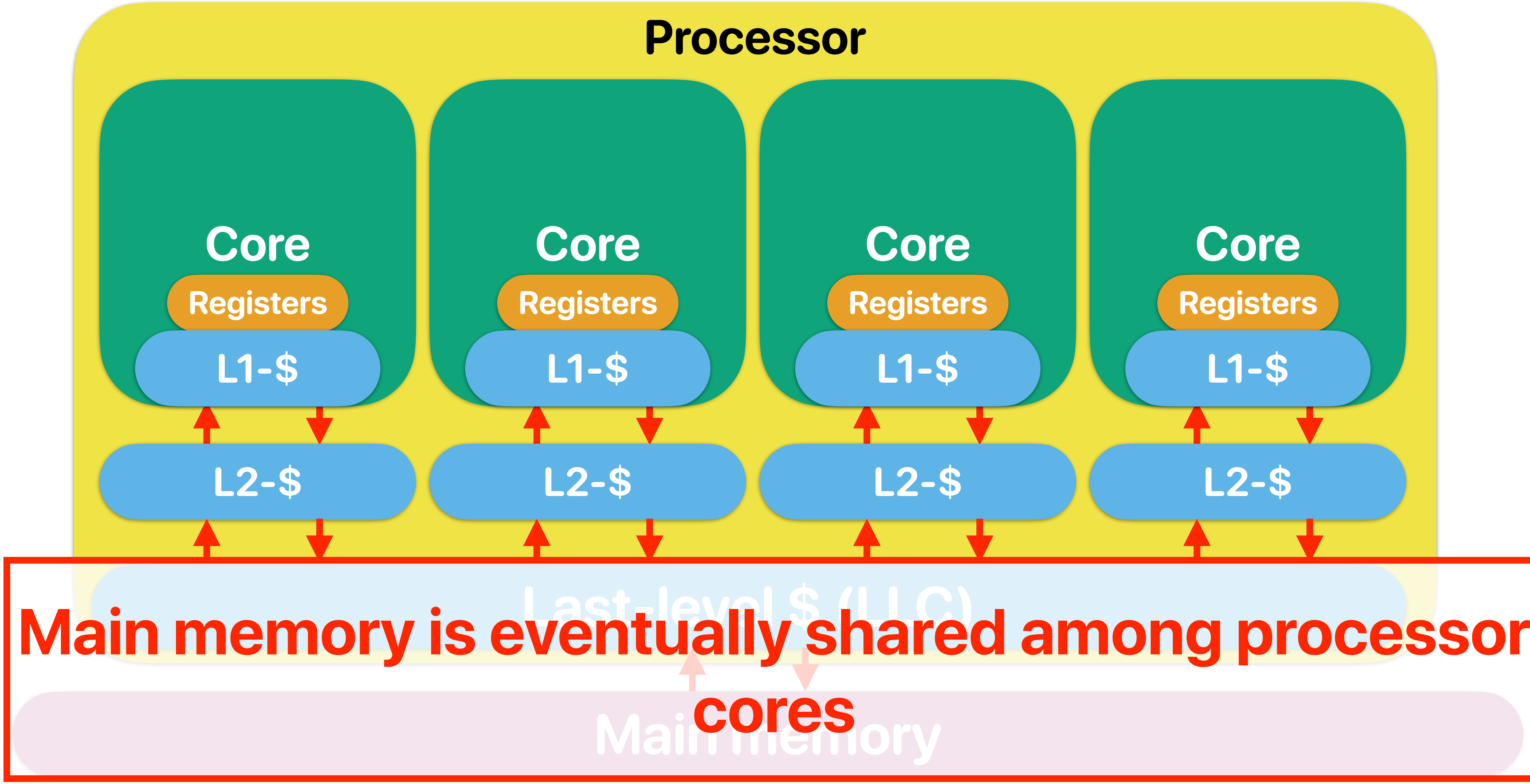
The cost of creating processes

- Measure process creation overhead using lmbench http://www.bitmover.com/lmbench/

Intel Sandy Bridge

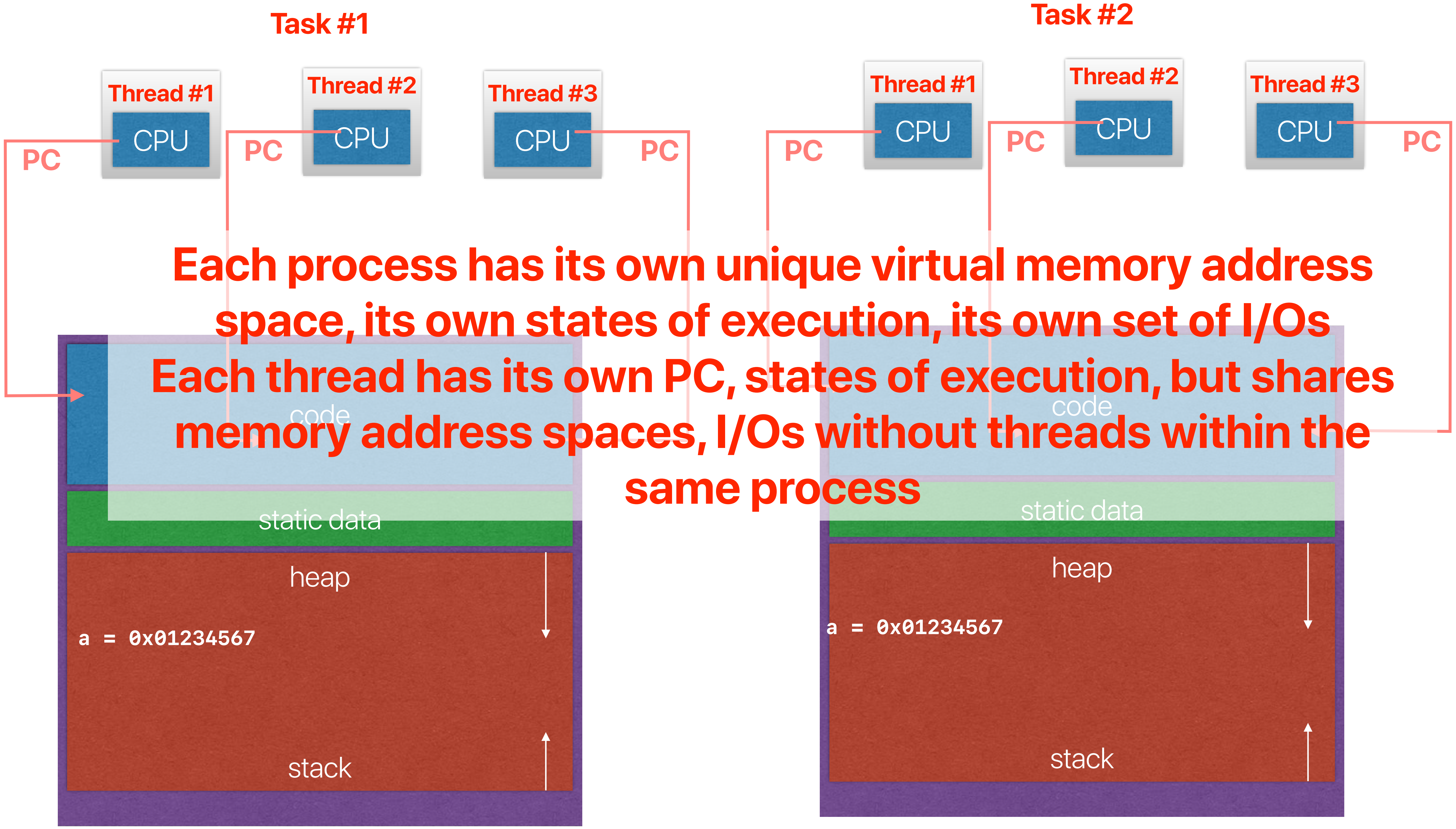


Concept of chip multiprocessors



Main memory is eventually shared among processor cores

Threads



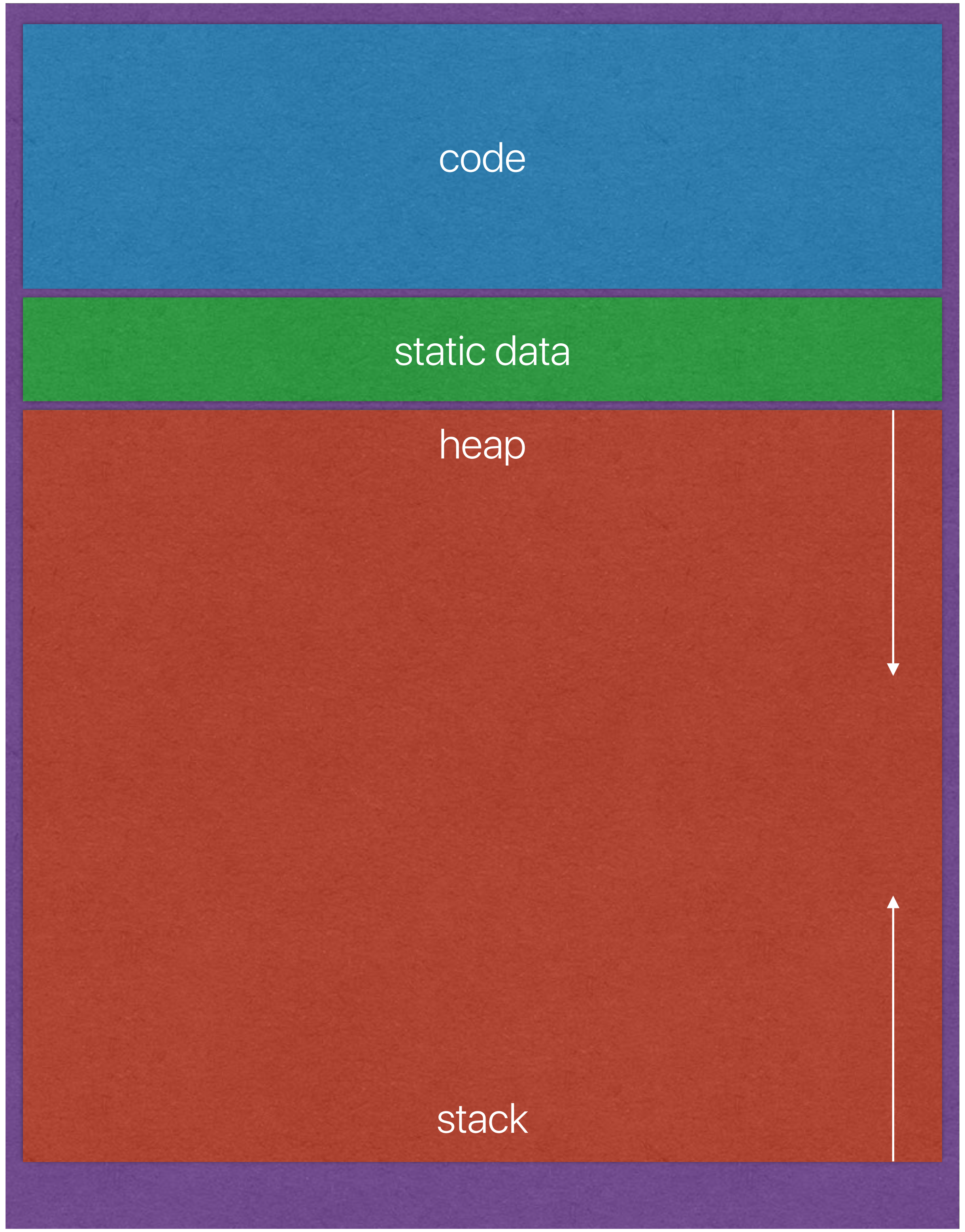
Each process has its own unique virtual memory address space, its own states of execution, its own set of I/Os

Each thread has its own PC, states of execution, but shares memory address spaces, I/Os without threads within the same process

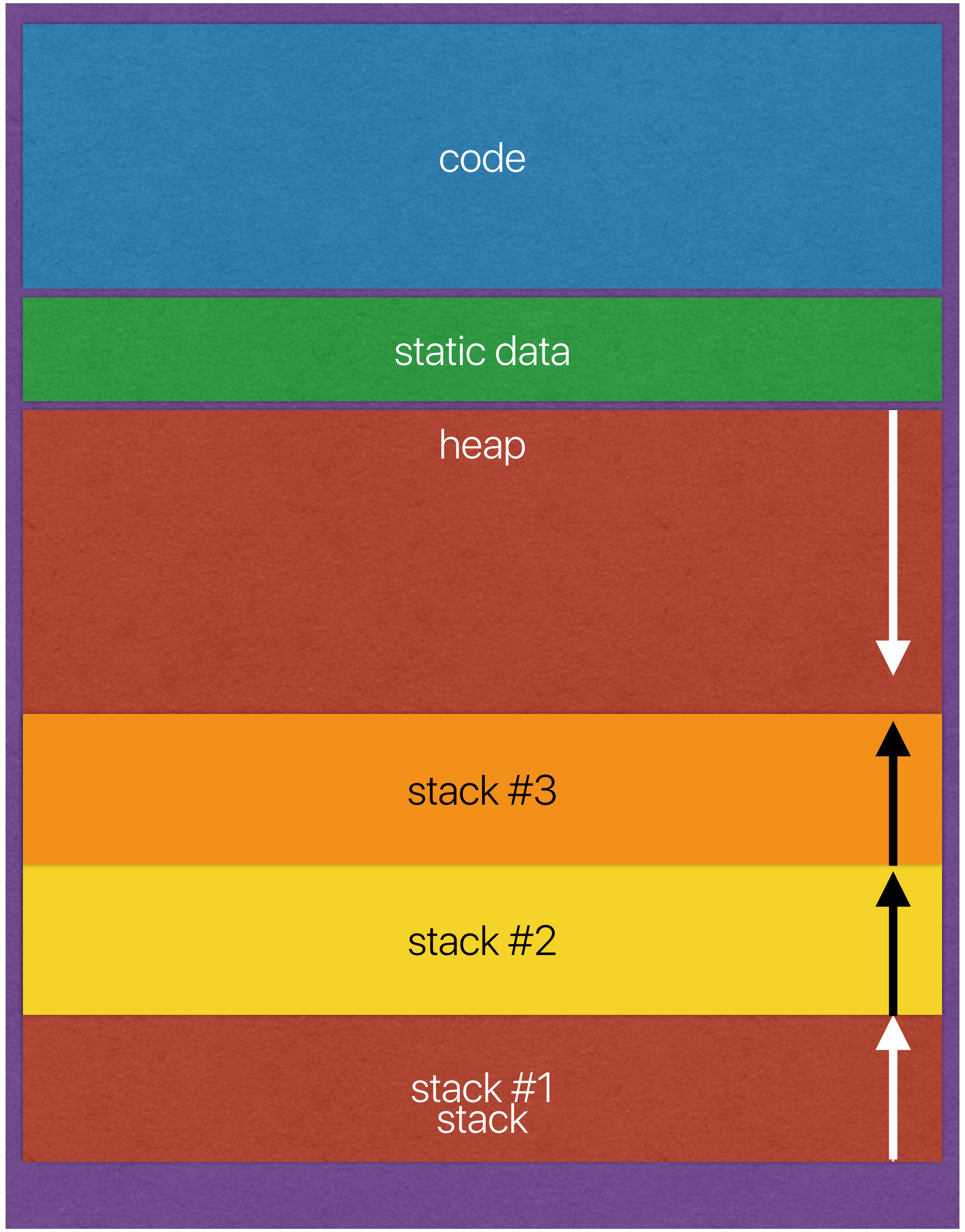
Why Threads?

- Process is an abstraction of a computer
  - When you create a process, you duplicate everything
  - However, you only need to duplicate CPU abstraction to parallelize computation tasks
- Threads as lightweight processes
  - Thread is an abstraction of a CPU in a computer
  - Maintain separate execution context
  - Share other resources (e.g. memory)

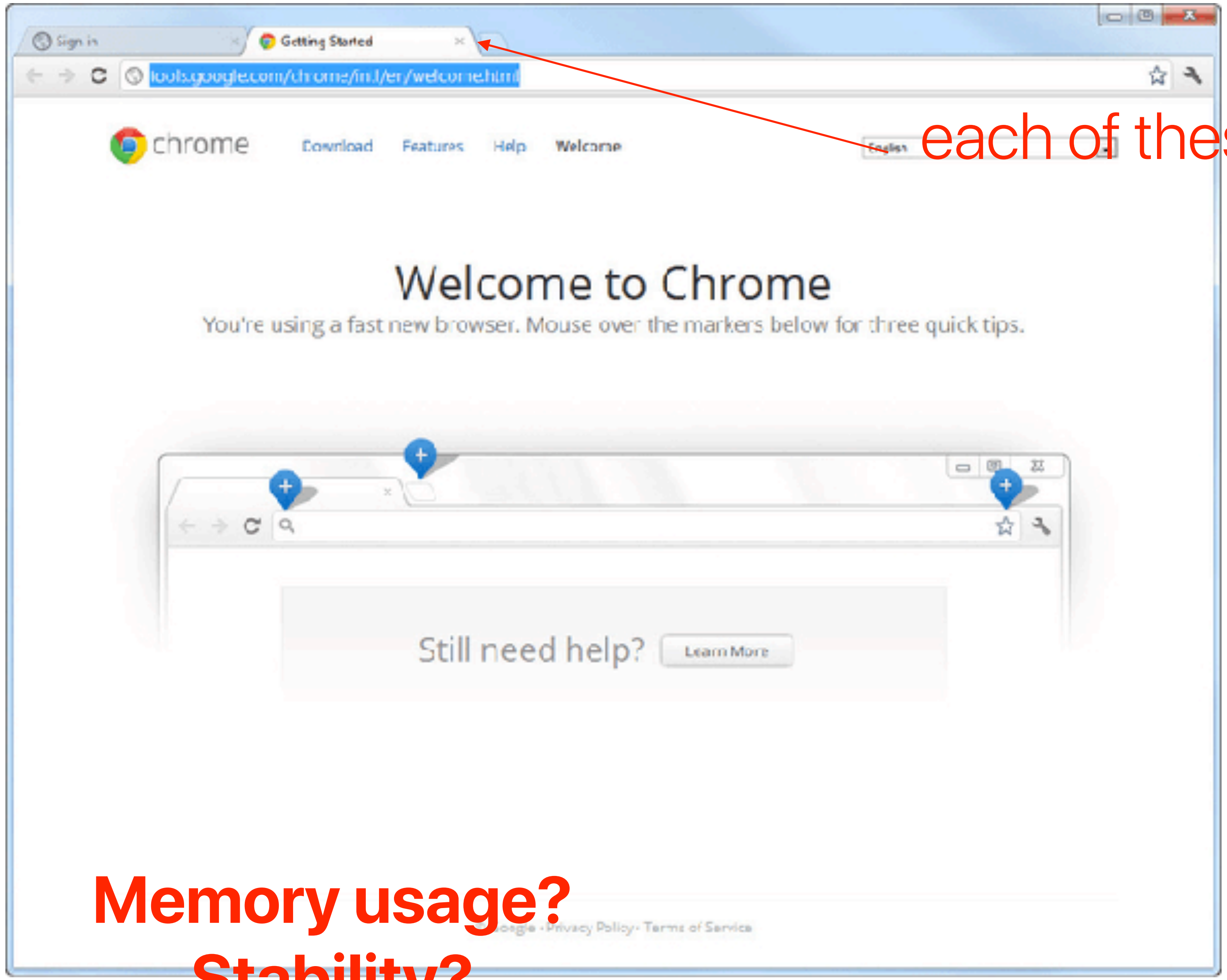
The virtual memory of single-threaded applications



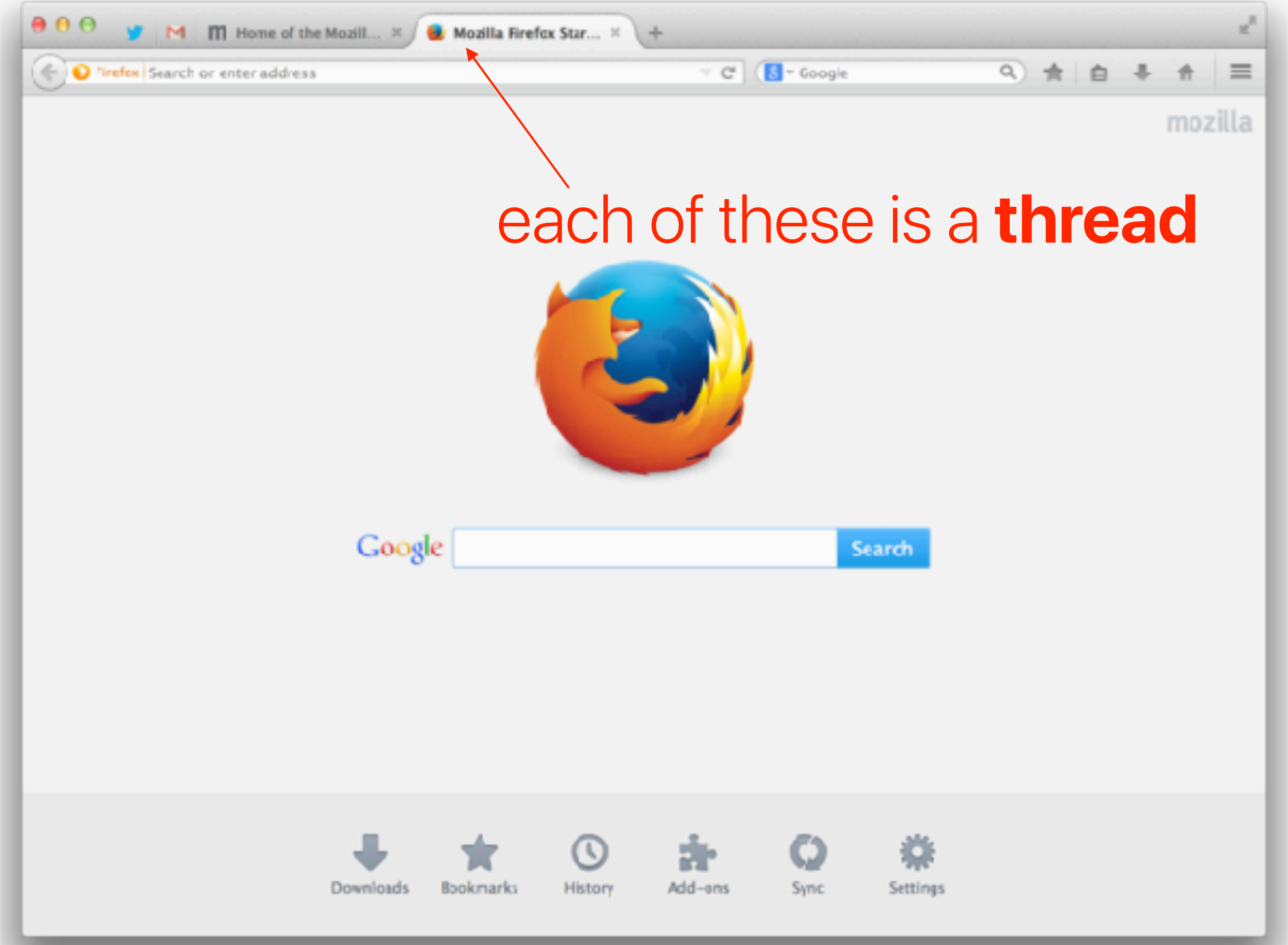
The virtual memory of multithreaded applications



Case study: Chrome v.s. Firefox



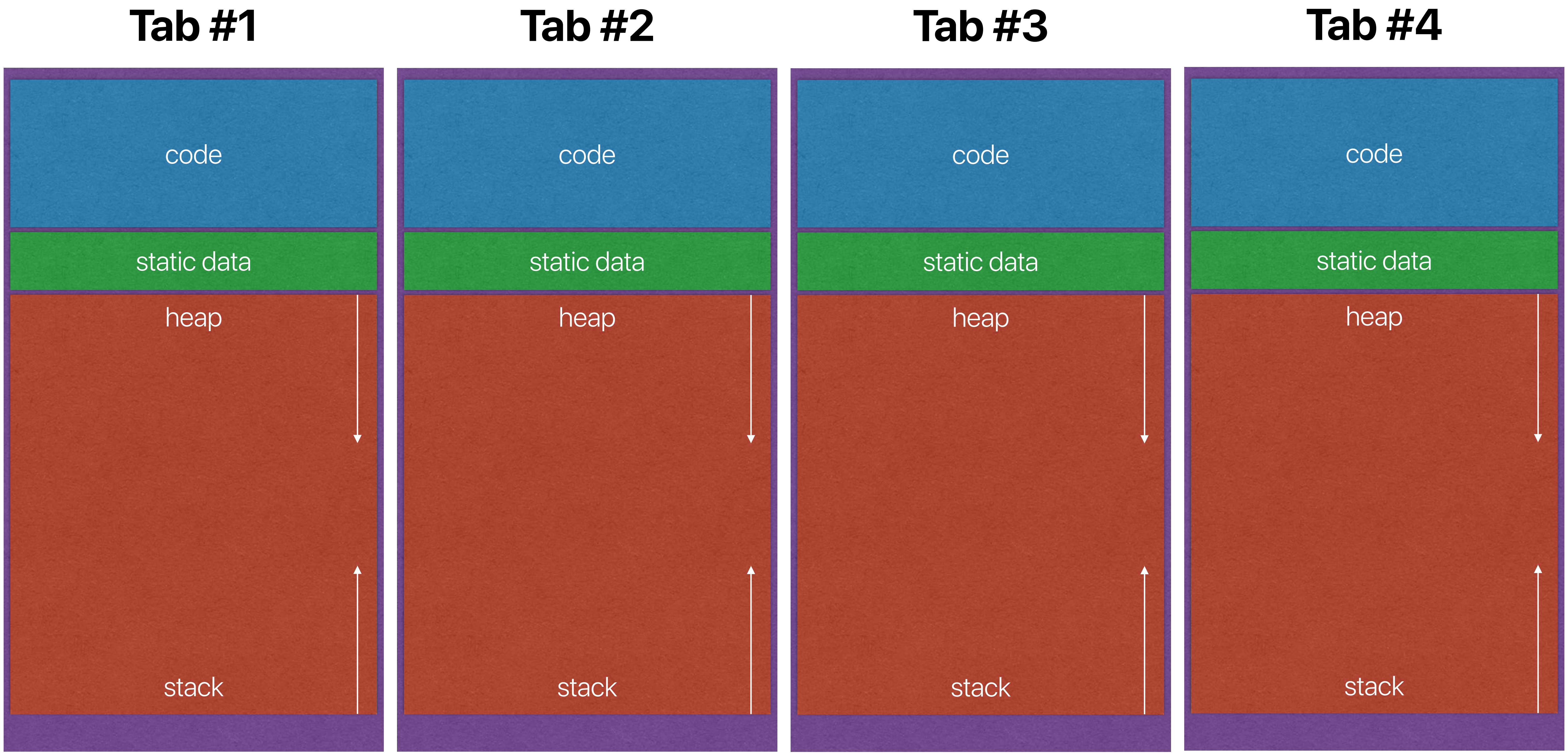
each of these is a **process**



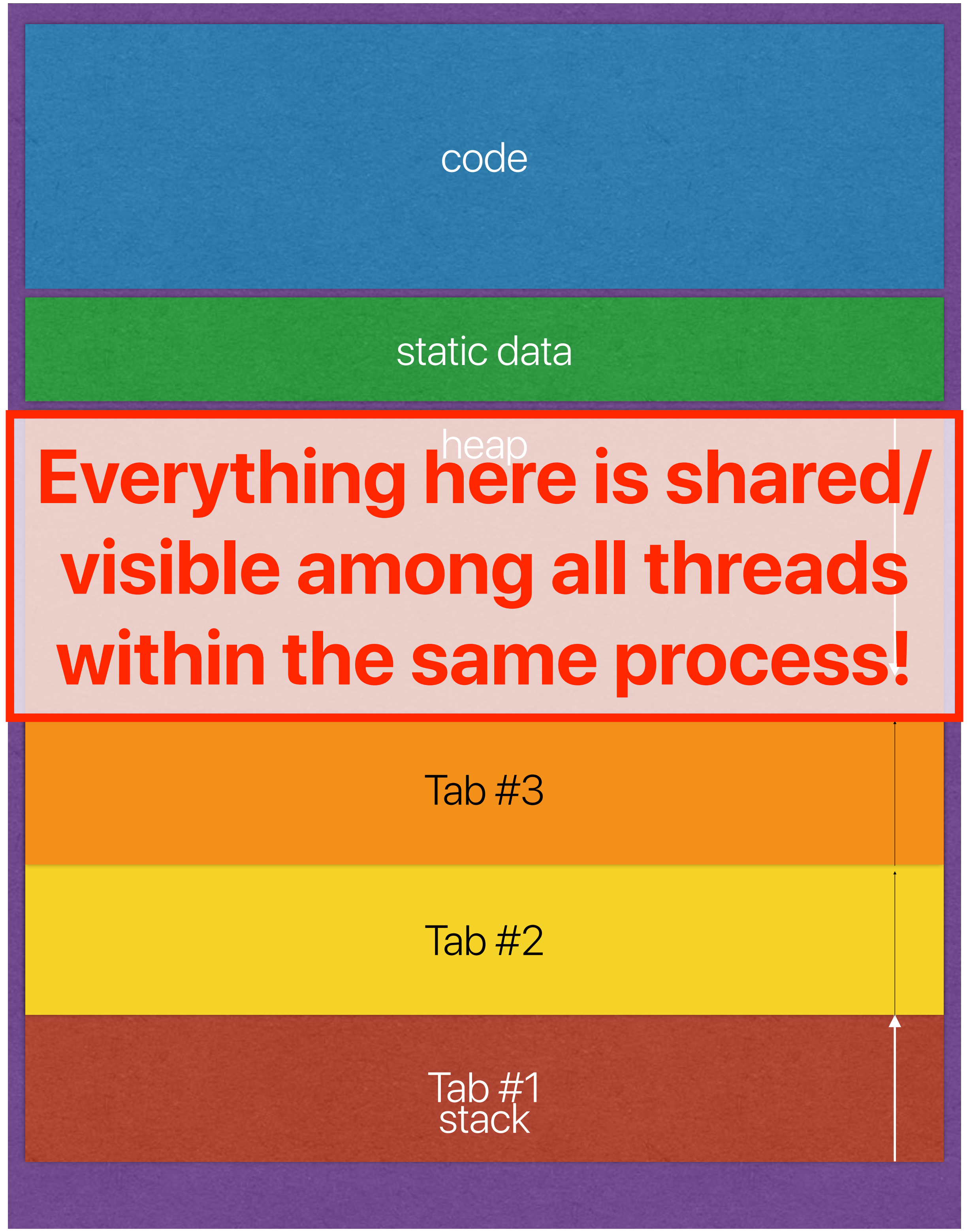
each of these is a **thread**

Memory usage?
Stability?
Security?
Latency?

Chrome


Tab #1
code
static data
heap
stack
Tab #2
code
static data
heap
stack
Tab #3
code
static data
heap
stack
Tab #4
code
static data
heap
stack

Firefox


code
static data
heap
Everything here is shared/
visible among all threads
within the same process!
Tab #3
Tab #2
Tab #1
stack

Why "Mach"?

- The hardware is changing
  - Multiprocessors
  - Networked computing

> be built and future development of UNIX-like systems for new architectures can continue. The computing environment for which Mach is targeted spans a wide class of systems, providing basic support for large, general purpose multiprocessors, smaller multiprocessor networks and individual workstations (see

- The software
  - The demand of extending an OS easily
  - Repetitive but confusing mechanisms for similar stuffs

> As the complexity of distributed environments and multiprocessor architectures increases, it becomes increasingly important to return to the original UNIX model of consistent interfaces to system facilities. Moreover, there is a clear need to allow the underlying system to be transparently extended to allow user-state processes to provide services which in the past could only be fully integrated into UNIX by adding code to the operating system kernel.

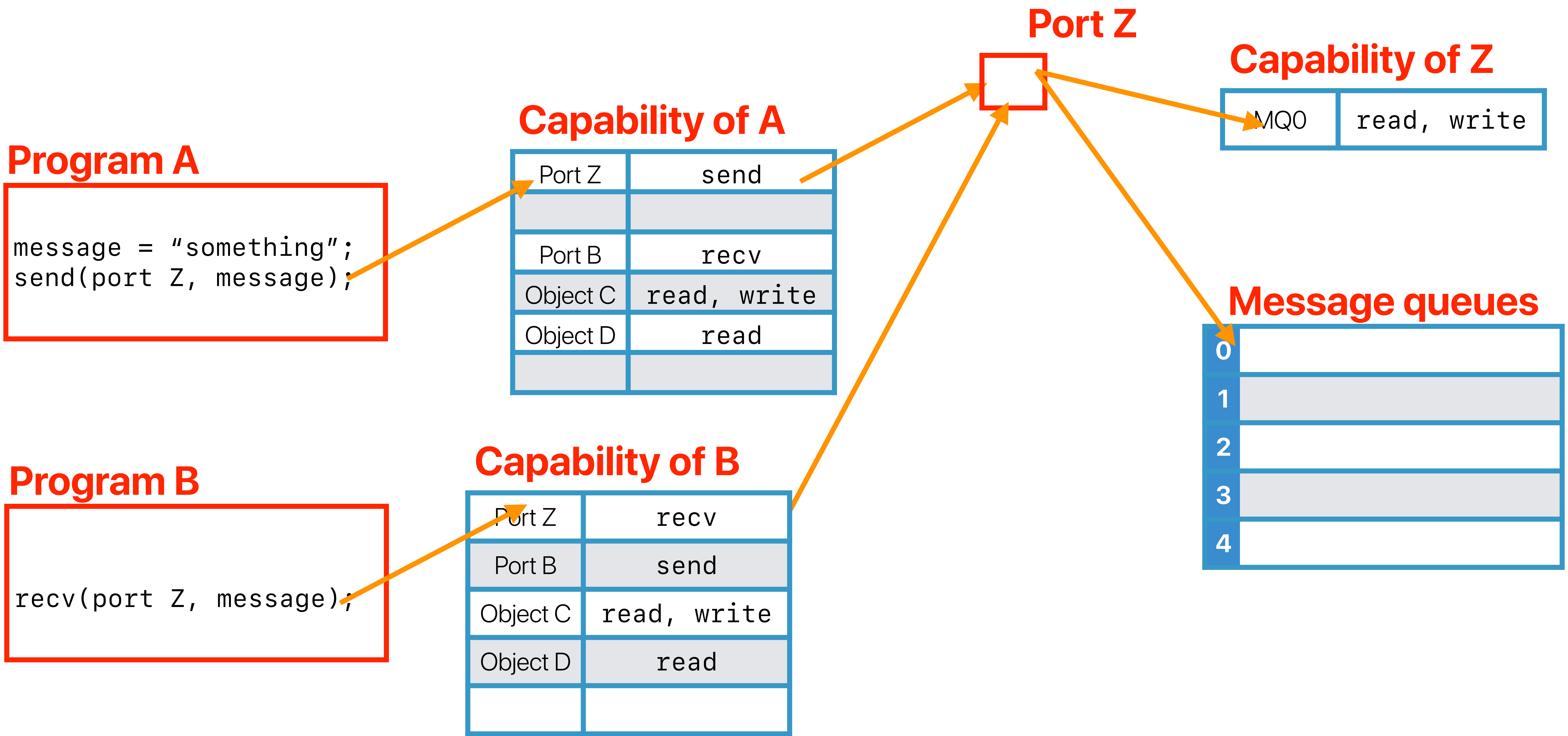
Interprocess communication

- UNIX provides a variety of mechanisms
  - Pipes
  - Pty's
  - Signals
  - Sockets
- No protection
- No consistency
- Location dependent

Ports/Messages

- Port is an abstraction of:
  - Message queues
  - Capability
- What do ports/messages promote?
  - Location independence — everything is communicating with ports/messages, no matter where it is

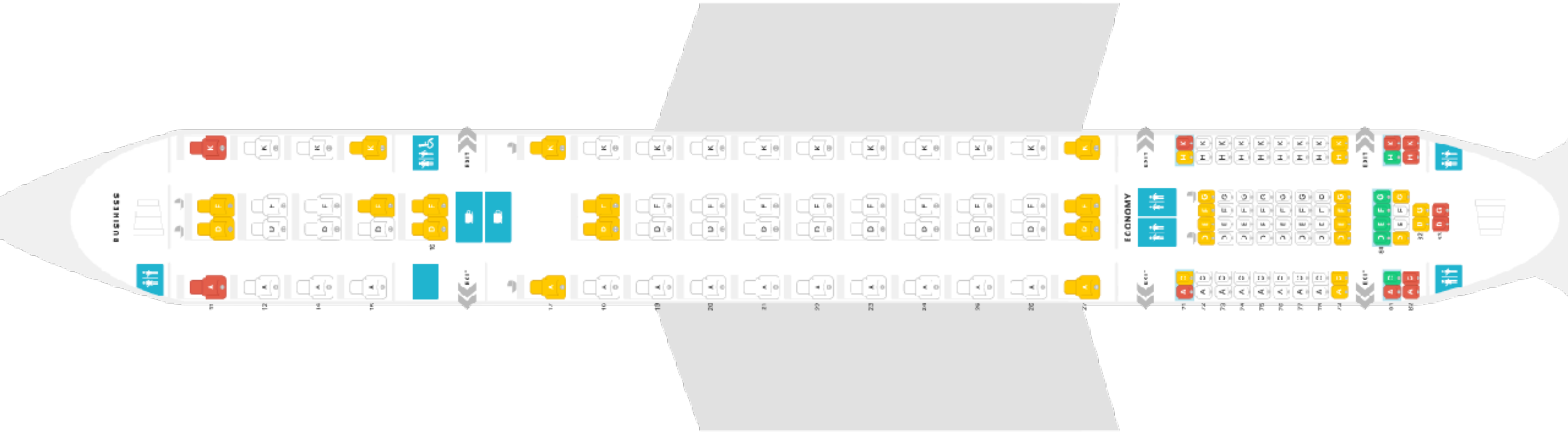
Ports/Messages



```java
class JBT {

    int variable = 5;

    public static void main(String args[]) {
        JBT obj = new JBT();

        obj.method(20);
        obj.method();
    }

    void method(int variable) {
        variable = 10;
        System.out.println("Value of Instance variable :" + this.variable);
        System.out.println("Value of Local variable :" + variable);
    }

    void method() {
        int variable = 40;
        System.out.println("Value of Instance variable :" + this.variable);
        System.out.println("Value of Local variable :" + variable);
    }
}
```

What is capability? — Hydra

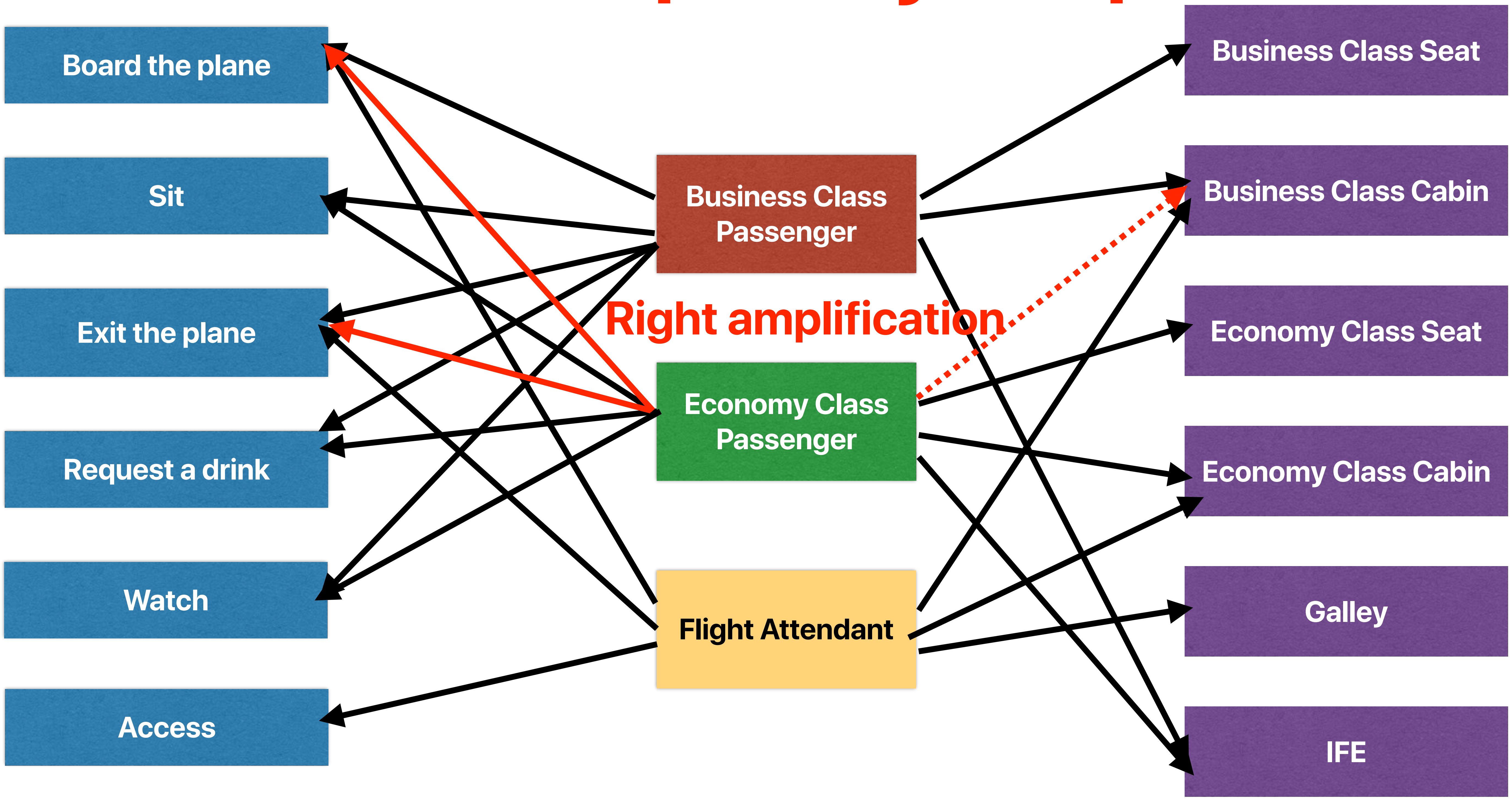
- An access control list associated with an object
- Contains the following:
  - A reference to an object
  - A list of access rights
- Whenever an operation is attempted:
  - The requester supplies a capability of referencing the requesting object — like presenting the boarding pass
  - The OS kernel examines the access rights
    - Type-independant rights
    - Type-dependent rights

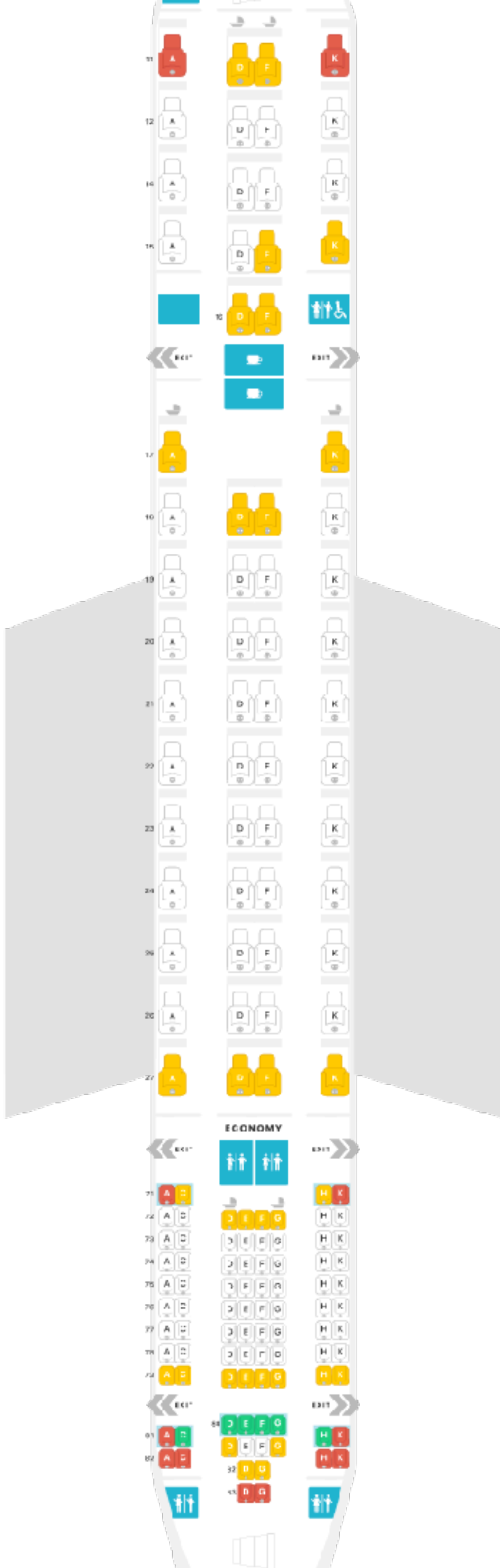

BUSINESS
ECONOMY

Capability v.s. boarding pass

- You can only enjoy the ground services (objects) that your booking class provides
- You can only access the facilities (objects) on the airplane according to the booking class

Capability in a plane





The impact of Mach

- Threads
- Extensible operating system kernel design
- Strongly influenced modern operating systems
  - Windows NT/2000/XP/7/8/10
  - MacOS

developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/Mach/Mach.html

Documentation Archive

Table of Contents

Mach Overview

The fundamental services and primitives of the OS X kernel are based on Mach 3.0. Apple has modified and extended Mach to better meet OS X functional and p

Mach 3.0 was originally conceived as a simple, extensible, communications microkernel. It is capable of running as a stand-alone kernel, with other traditional o networking stacks running as user-mode servers.

However, in OS X, Mach is linked with other kernel components into a single kernel address space. This is primarily for performance; it is much faster to make a messages or do remote procedure calls (*RPC*) between separate tasks. This modular structure results in a more robust and extensible system than a monolithic l microkernel.

Thus in OS X, Mach is not primarily a communication hub between clients and servers. Instead, its value consists of its abstractions, its extensibility, and its flex

- object-based APIs with communication channels (for example, ports) as object references
- highly parallel execution, including preemptively scheduled threads and support for *SMP*
- a flexible scheduling framework, with support for real-time usage
- a complete set of *IPC* primitives, including messaging, *RPC*, synchronization, and notification
- support for large virtual address spaces, shared memory regions, and memory objects backed by persistent store
- proven extensibility and portability, for example across instruction set architectures and in distributed environments
- security and resource management as a fundamental principle of design; all resources are virtualized

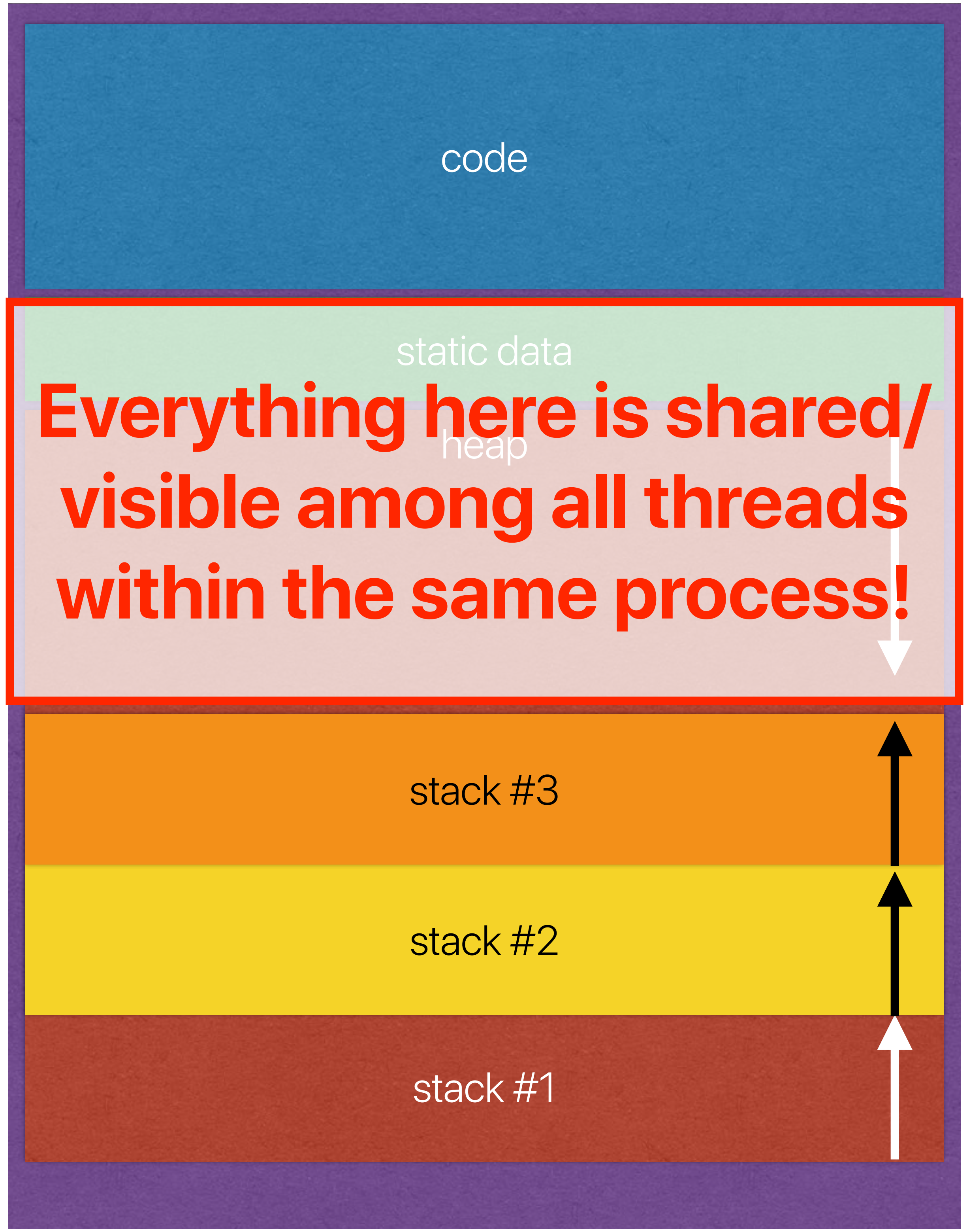
Mach Kernel Abstractions

Mach provides a small set of abstractions that have been designed to be both simple and powerful. These are the main kernel abstractions:

- *Tasks*. The units of resource ownership; each task consists of a virtual address space, a *port right namespace*, and one or more *threads*. (Similar to a process.
- *Threads*. The units of CPU execution within a task.
- *Address space*. In conjunction with memory managers, Mach implements the notion of a sparse virtual address space and shared memory.
- *Memory objects*. The internal units of memory management. Memory objects include named entries and regions; they are representations of potentially persi
- *Ports*. Secure, simplex communication channels, accessible only via send and receive capabilities (known as port rights).
- *IPC*. Message queues, remote procedure calls, notifications, semaphores, and lock sets.
- *Time*. Clocks, timers, and waiting.

Thread programming & synchronization

The virtual memory of multithreaded applications



Joint Banking

withdraw
$20

Bank of UCR

What is the new balance each would see?

deposit
$10

current balance: $40

Processors/threads are not-deterministic

- Processor/compiler may reorder your memory operations/ instructions
- Each processor core may not run at the same speed (cache misses, branch mis-prediction, I/O, voltage scaling and etc..)
- Threads may not be executed/scheduled right after it’s spawned

Synchronization

- Concurrency leads to multiple active processes/threads that share one or more resources
- Synchronization involves the orderly sharing of resources
- All threads must be on the same page
- Need to avoid race conditions

Critical sections

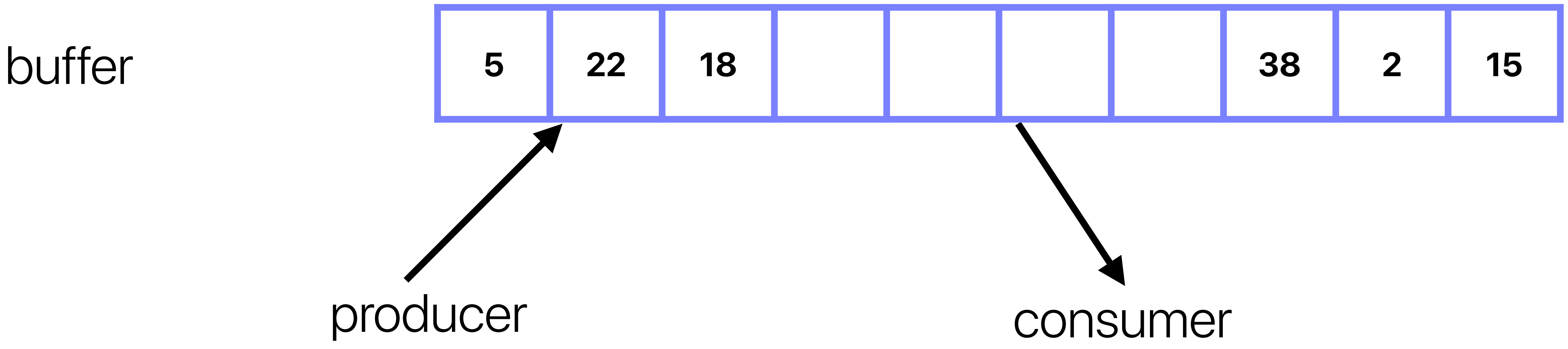
- Protect some pieces of code that access shared resources (memory, device, etc.)
- For safety, critical sections should:
  - Enforce mutual exclusion (i.e. only one thread at a time)
  - Execute atomically (all-or-nothing) before allowing another thread

Solving the “Critical Section Problem”

1. Mutual exclusion — at most one process/thread in its critical section
2. Progress — a thread outside of its critical section cannot block another thread from entering its critical section
3. Fairness — a thread cannot be postponed indefinitely from entering its critical section
4. Accommodate nondeterminism — the solution should work regardless the speed of executing threads and the number of processors

Bounded-Buffer Problem

- Also referred to as "producer-consumer" problem
- Producer places items in shared buffer
- Consumer removes items from shared buffer



Without synchronization, you may write

```
int buffer[BUFF_SIZE]; // shared global
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        int item = buffer[out];
        out = (out + 1) %
BUFF_SIZE;
        // do something w/ item
    }
    return NULL;
}
```

Use locks

```c
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```c
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

```c
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

How to implement lock/unlock

The baseline

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
  fprintf(stderr, "usage: main-first
<loopcount>\n");
  exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %d] [%x]\n", \
balance, (unsigned int) &balance);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %d]\n [should:
%d]\n",
     balance, max*2);
    return 0;
}
```

```
int max;
volatile int balance = 0; // shared global varia
```

```
void * mythread(void *arg) {
    char *letter = arg;
    int i; // stack (private per
thread)
    printf("%s: begin [addr of i:
%p]\n", letter, &i);
    for (i = 0; i < max; i++) {
        balance = balance + 1; //
shared: only one
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

Use pthread_lock

```
int max;
volatile int balance = 0; // shared global variable
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
  fprintf(stderr, "usage: main-first <loopcount>\n");
  exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %d] [%x]\n", balance,
     (unsigned int) &balance);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %d]\n [should: %d]\n",
     balance, max*2);
    return 0;
}
```

```
void * mythread(void *arg) {
    char *letter = arg;
    printf("%s: begin\n", letter);
    int i;
    for (i = 0; i < max; i++) {
      Pthread_mutex_lock(&lock);
      balance++;
      Pthread_mutex_unlock(&lock);
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

Use pthread_lock (coarser grain)

```c
int max;
volatile int balance = 0; // shared global variable
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

```c
int main(int argc, char *argv[])
{
    if (argc != 2) {
  fprintf(stderr, "usage: main-first <loopcount>\n");
  exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %d] [%x]\n", balance,
      (unsigned int) &balance);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %d]\n [should: %d]\n",
      balance, max*2);
    return 0;
}
```

```c
void * mythread(void *arg) {
    char *letter = arg;
    printf("%s: begin\n", letter);
    int i;
      Pthread_mutex_lock(&lock);
      for (i = 0; i < max; i++) {
         balance++;
      }
      Pthread_mutex_unlock(&lock);
    printf("%s: done\n", letter);
    return NULL;
}
```

Use spin locks

```c
int max;
volatile int balance = 0; // shared global variable
volatile unsigned int lock = 0;
```

```c
void * mythread(void *arg) {
    char *letter = arg;
    printf("%s: begin\n", letter);
    int i;
    for (i = 0; i < max; i++) {
      SpinLock(&lock);
      balance++;
      SpinUnlock(&lock);
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

```c
int main(int argc, char *argv[])
{
    if (argc != 2) {
   fprintf(stderr, "usage: main-first <loopcount>\n");
   exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %d] [%x]\n", balance,
      (unsigned int) &balance);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %d]\n [should: %d]\n",
      balance, max*2);
    return 0;
}
```

what if context switch happens here?

```c
void SpinLock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    // spin
    *lock = 1;          // SET (lock)
}

void SpinUnlock(volatile unsigned int *lock)
    *lock = 0;
}
```

Use spin locks

```c
int max;
volatile int balance = 0; // shared global variable
volatile unsigned int lock = 0;
```

```c
void * mythread(void *arg) {
    char *letter = arg;
    printf("%s: begin\n", letter);
    int i;
    for (i = 0; i < max; i++) {
      SpinLock(&lock);
      balance++;
      SpinUnlock(&lock);
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

```c
int main(int argc, char *argv[])
{
    if (argc != 2) {
   fprintf(stderr, "usage: main-first <loopcount>\n");
   exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %d] [%x]\n", balance,
      (unsigned int) &balance);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %d]\n [should: %d]\n",
      balance, max*2);
    return 0;
}
```

```c
void SpinLock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    // spin
    *lock = 1;         // SET (lock)
}

void SpinUnlock(volatile unsigned int *lock) {
    *lock = 0;
}
```

what if context switch happens here?

— the lock must be updated atomically

Use spin locks

```
int max;
volatile int balance = 0; // shared global variable
volatile unsigned int lock = 0;
```

```
void * mythread(void *arg) {
    char *letter = arg;
    printf("%s: begin\n", letter);
    int i;
    for (i = 0; i < max; i++) {
```

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
  fprintf(stderr, "usage: main-first
  exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %d
     (unsigned int) &balance);
    Pthread_create(&p1, NULL, mythrea
    Pthread_create(&p2, NULL, mythrea
    // join waits for the threads to
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %d
      balance, max*2);
    return 0;
}
```

```
static inline uint xchg(volatile unsigned int *addr,
unsigned int newval) {
    uint result;
    asm volatile("lock; xchgl %0, %1" : "+m" (*addr),
"=a" (result) : "1" (newval) : "cc");
    return result;
}

void SpinLock(volatile unsigned int *lock) {
    // what code should go here?
}

void SpinUnlock(volatile unsigned int *lock) {
    // what code should go here?
}
```

exchange the content in %0 and %1

a prefix to xchgl that locks the whole cache line

Use spin locks

```
int max;
volatile int balance = 0; // shared global variable
volatile unsigned int lock = 0;
```

```
void * mythread(void *arg) {
    char *letter = arg;
    printf("%s: begin\n", letter);
    int i;
    for (i = 0; i < max; i++) {
```

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
  fprintf(stderr, "usage: main-first
  exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %c
     (unsigned int) &balance);
    Pthread_create(&p1, NULL, mythrea
    Pthread_create(&p2, NULL, mythrea
    // join waits for the threads to
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %c
     balance, max*2);
    return 0;
}
```

```
static inline uint xchg(volatile unsigned int *addr, unsigned
int newval) {
    uint result;
    asm volatile("lock; xchgl %0, %1" : "+m" (*addr),
"=a" (result) : "1" (newval) : "cc");
    return result;
}

void SpinLock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1);
}

void SpinUnlock(volatile unsigned int *lock) {
    xchg(lock, 0);
}
```

Semaphores

Semaphores

- A synchronization variable
- Has an integer value — current value dictates if thread/process can proceed
- Access granted if val > 0, blocked if val == 0
- Maintain a list of waiting processes

Semaphore Operations

- `sem_wait(S)`
  - if S > 0, thread/process proceeds and decrement S
  - if S == 0, thread goes into "waiting" state and placed in a special queue
- `sem_post(S)`
  - if no one waiting for entry (i.e. waiting queue is empty), increment S
  - otherwise, allow one thread in queue to proceed

Semaphore Op Implementations

```
sem_init(sem_t *s, int initvalue) {
    s->value = initvalue;
}
```

```
sem_wait(sem_t *s) {
    while (s->value <= 0)
        put_self_to_sleep(); // put self to sleep
    s->value--;
}
```

```
sem_post(sem_t *s) {
    s->value++;
    wake_one_waiting_thread(); // if there is one
}
```

Atomicity in Semaphore Ops

- Semaphore operations must operate atomically
  - Requires lower-level synchronization methods requires (test-and-set, etc.)
  - Most implementations still require on busy waiting in spinlocks
- What did we gain by using semaphores?
  - Easier for programmers
  - Busy waiting time is limited